EcoAndroid: An Android Studio Plugin for Developing Energy-Efficient Java Mobile Applications

Ana Ribeiro* and João F. Ferreira†
INESC-ID & Instituto Superior Técnico
University of Lisbon, Lisbon, Portugal
Email: *anasofiaribeiro@tecnico.ulisboa.pt, †joao@joaoff.com

Alexandra Mendes
University of Beira Interior, Covilhã, Portugal
HASLab, INESC TEC, Porto, Portugal
Email: alexandra@archimendes.com

***Abstract*—Mobile devices have become indispensable in our daily life and reducing the energy consumed by them has become essential. However, developing energy-efficient mobile applications is not a trivial task. To address this problem, we present EcoAndroid, an Android Studio plugin that automatically applies energy patterns to Java source code. It currently supports ten different cases of energy-related refactorings, divided over five energy patterns taken from the literature. We used EcoAndroid to analyze 100 Java mobile applications (≈ 1.5M LOC) and we found that 35 of the projects had a total of 95 energy code smells. EcoAndroid was able to automatically refactor all the code smells identified.**

***Index Terms*—Green Software, Energy Consumption, Energy Patterns, Code Smells, Android, Refactoring**

I. Introduction

Mobile devices have become a fundamental accessory in current day-to-day life. They are used as credit cards, work tools, educational helpers, among many other useful purposes. Unfortunately, the battery power on them is finite and, despite the advances in hardware and battery technology, the needs of most users are not yet met. Indeed, Wilke et al. [1] analyzed comments left in the Google Play market place for Android applications and concluded that 18% of the complaints were related to energy problems. The reduction of the energy consumed by mobile devices has thus become an important non-functional requirement [2], [3].

One way of decreasing the energy consumed by a mobile device is to ensure that the mobile applications that the device runs are energy-efficient. However, this is a complex task since a lot of factors can influence energy consumption, such as: the mobile networking technology used (3G, GSM or WiFi) [4]; heavy graphic processing; and screen usage while on an application. Taking these factors into consideration is not always trivial, as they can be easily overlooked by developers when coding.

An approach that makes the development of energy-efficient mobile applications easier is following so-called *energy patterns*, which are code patterns known to use energy prudently. Work documenting these patterns has been growing in recent years [5]–[8]. An extremely useful resource is Cruz and Abreu's catalog of energy-related patterns [5]. This catalog can be of great assistance to mobile application developers, as it describes each pattern and its context, also providing a series of examples and references. However, *the manual application of these patterns is far from trivial and can be time-consuming*.

To address this problem, we present EcoAndroid, an Android Studio plugin that automatically applies a set of energy patterns to Java source code. At the time of writing, the plugin supports ten different cases of energy-related refactorings, distributed over five energy patterns taken from the literature [5].

We used EcoAndroid to analyze 100 Java projects (≈ 1.5M LOC) from F-Droid, a Free and Open Source Android App Repository, and we found that 35 of the projects had a total of 95 energy code smells detected by the plugin. EcoAndroid was able to automatically refactor all the code smells identified.

EcoAndroid is an extendable Android Studio plugin, created to assist developers in creating energy-efficient mobile applications. Since it is open source[1], it can also be extended by the research community to explore new techniques that help in the creation of more energy-efficient mobile applications. EcoAndroid is currently available in the JetBrains Marketplace: https://plugins.jetbrains.com/plugin/15637-ecoandroid

Illustrative Example: To illustrate the type of refactoring that EcoAndroid is capable of, we show in Figure 1 an example of a refactoring suggested and applied by EcoAndroid. It consists in the application of the *Cache - Check Metadata* energy pattern in the Android mobile application Taskbar[2], which puts a start menu and recent apps tray on top of the screen that is accessible at any time. Taskbar is a popular application: at the time of writing, its GitHub project has 405 stars and it has been downloaded more than 1 million times from the app store Google Play. Figure 1a shows the original source code, where the code smell was detected. There is an opportunity to optimize the energy efficiency of the code by caching the object bundle and only executing the code if the object has changed. Figure 1b shows the source code after the refactoring automatically performed by EcoAndroid.

[1]EcoAndroid Github Repository: https://github.com/sr-lab/EcoAndroid

[2]Taskbar (Google Play): https://play.google.com/store/apps/details?id=com.farmerbb.taskbar.

```
public final class TaskerConditionReceiver
                        extends BroadcastReceiver {
 @Override
 public void onReceive(Context context, Intent intent)
 {
  if(U.isExternalAccessDisabled(context)) return;
  BundleScrubber.scrub(intent);
  final Bundle bundle =
   intent.getBundleExtra(com.twofortyfouram.locale.
                            api.Intent.EXTRA_BUNDLE);
   ...
   if(PluginBundleManager.isBundleValid(bundle))
   { ... }
   ...
 }
}
```

(a) Original code: code smell detected.

```
public final class TaskerConditionReceiver
                        extends BroadcastReceiver {
 private Bundle lastbundle = null;

 @Override
 public void onReceive(Context context, Intent intent)
 {
  if(U.isExternalAccessDisabled(context)) return;
  if(lastbundle.equals(intent.getBundleExtra(com.
   twofortyfouram.locale.api.Intent.EXTRA_BUNDLE)))
  {
    // bundle hasn't changed: we can safely return
    return;
  }
  updateValues(intent);
  ...
  if(PluginBundleManager.isBundleValid(lastbundle))
  { ... }
  ...
 }

 private void updateValues(Intent intent) {
   lastbundle =
    intent.getBundleExtra(com.twofortyfouram.locale.
                            api.Intent.EXTRA_BUNDLE);
 }
}
```

(b) Refactored code: energy pattern applied.

Fig. 1: Illustrative example of an EcoAndroid refactoring applying the energy pattern *Cache - Check Metadata* on the popular application Taskbar.

II. Background and Related Work

This section presents an overview of the main concepts used in this paper and discusses related work.

A. Energy Patterns

An *Energy Pattern* can be defined as a design pattern that mobile application developers can adopt to improve the energy efficiency of mobile applications. It is common to present energy patterns with an anti-pattern example and a description of the alterations needed to reduce the energy consumed. Studies with the goal of documenting energy patterns have emerged over recent years [5]–[9].

B. Mobile Applications Environments and Languages

A study done by Habchi et al. [10] compared the ratio of energy code smells in iOS and Android mobile applications, concluding that the latter had a higher number of energy code smells. The study also states that these differences are related to the platform and not to the difference in programming languages. The main languages for programming iOS mobile applications are Swift and Objective-C while for Android mobile applications are Java and Kotlin. In terms of IDEs, Android Studio (built on IntelliJ), IntelliJ, and Eclipse are popular choices, being that the first one is the official one for Android development and the one chosen for this project. Note that, even though we focus on Android mobile applications, Android Studio can also be used to develop iOS mobile applications. As long as these applications are written in the Java programming language, EcoAndroid can be used.

C. Energy-Related Refactoring for Java Source Code

This subsection focuses on refactoring tools that optimize the energy efficiency of Java mobile applications. Table I lists the tools and their respective environment.

Leafactor [11], [12] is a tool that automatically refactors Android mobile applications source code to reduce energy consumption. Leafactor is an Eclipse plugin while EcoAndroid is compatible with both Android Studio and IntelliJ. The 5 refactorings supported by Leafactor are acquired from a previous study by the same authors about the effect of performance-based practices on mobile applications' energy consumption [8]. **Chimera** [13] covers 11 energy-greedy code patterns. It uses Lint[3] for the inspection phase and Autorefactor [14] for the refactoring phase. A new aspect about this project is how broad the evaluation is, inspecting more than 600 mobile applications. In their paper, Couto et al. [13] also compare the energy savings of combinations of refactorings. **AEON** (Automated Android Energy-Efficiency Inspection) [15] is a support framework, compatible with IntelliJ and Android Studio. It automatically detects energy inefficiencies in Android mobile applications and helps developers fix those inefficiencies. It also supports developers in verifying, refactoring and profiling such inefficiencies. **EARMO** [16] is an approach that detects and corrects energy-related anti-patterns in mobile applications, while accounting for energy consumption when performing the refactorings. It supports 8 anti-patterns within two categories: Object-oriented specific and Android-specific. The refactoring is achieved via refactoring-tool-support provided by Android Studio and

[3]*Lint* is a code analysis tool (developer.android.com/studio/write/lint).

Tool/Approach Name	Environment
Leafactor [11], [12]	Eclipse
Chimera [13]	-
AEON [15]	IntelliJ, Android Studio
EARMO [16]	IntelliJ, Manually
aDoctor [17], [22]	Eclipse, Android Studio
Greeness category [19]	Android Studio (Android lint)
HOT-PEPPER [20]	-

TABLE I: Energy-Specific Refactoring Tools.

Eclipse. When that is not possible, the changes have to be applied manually. Palomba et al. propose **aDoctor** [17], a tool that identifies 15 Android-specific code smells from a catalog by Reimann et al. [18]. It is built on top of the Eclipse Java Development Toolkit (JDK). *aDoctor* was also extended as an Android Studio plugin supporting 5 energy-related refactorings. Le Goaër presents a new category in Android lint entitled **Greenness** [19]. This category has 11 checks, which can be viewed as an inspection in Android Studio. **HOT-PEPPER** [20] is able to detect and correct 3 types of Android-specific code smells. It uses PAPRIKA [21], a static analysis tool for Android apps for the detection and correction of code smells. As a final step, HOT-PEPPER uses a tool called NAGA VIPER, to compute energy metrics and evaluate the impact of corrected APKs, being able to inform the developer which APK is the best energy-efficient version for a given scenario.

When compared to the tools above, EcoAndroid is the first to support refactorings associated with the energy patterns *Dynamic Retry Delay*, *Push Over Poll*, *Cache*, and *Avoid Extraneous Graphics and Animations*.

III. EcoAndroid: Architecture and Implementation

This section describes the architecture of EcoAndroid and the cases that were implemented.

A. Overview

EcoAndroid is an Android Studio plugin that suggests automated refactorings with the aim of reducing energy consumption of Java android applications. Android Studio is an IDE built on JetBrains' IntelliJ IDEA. Due to this fact, the EcoAndroid plugin is also compatible with IntelliJ. Android Studio was chosen because it is the official IDE for Android app development, making it the best choice for maximizing the impact of our work.

To the best of our knowledge, there are no general-purpose refactoring plugins for Android Studio that can serve as the basis for this project. Thus, to aid in the refactoring of the source code, the *Program Stucture Interface* (PSI) of IntelliJ was used [23]. PSI is a layer of the IntelliJ Platform responsible for parsing files and creating the syntactic and

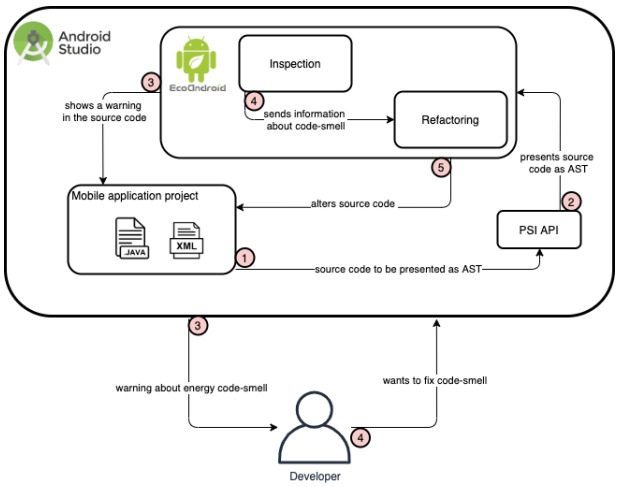

Fig. 2: EcoAndroid detection and refactoring process.

semantic code model. It creates PSI files, that are the root of a structure representing the contents of a file as a hierarchy of elements in a particular programming language. PSI is a read-write representation of the source code as a tree of elements corresponding to the structure of a source file. The PSI can be modified by adding, replacing and deleting PSI elements. These features allow the detection of possible energy improvements and the refactoring itself.

An IntelliJ's plugin can have two types of inspections: a *local inspection* or a *global inspection*. As the names suggest, a local inspection looks at only one file, while a global inspection looks at a group of files. Due to this, a global inspection does not show warnings along the source code but needs to be run manually by the user. Since we want EcoAndroid to be a tool that developers use interactively during development, every case that the plugin supports is implemented as a local inspection.[4] The plugin supports a total of five energy patterns, divided in ten separated cases. Each case is implemented as a local inspection in the plugin.

Figure 2 illustrates the process flow of the user interaction between the developer and EcoAndroid. The plugin starts by performing a static analysis, aided by the PSI API. The source code is represented as an Abstract Syntax Tree (AST) (actions (1) and (2)). If a code smell is found, a warning is shown to the developer (action (3)) and, if the they wish to do so (action (4)), the refactoring, which is also aided by the PSI API, is executed (action (5)).

B. Cases Implemented

The five energy patterns supported are a subset of the ones presented in the catalog by Cruz and Abreu [5]. The energy patterns are: *Dynamic Retry Delay*, *Push Over Poll*, *Reduce Size*, *Cache*, and *Avoid Extraneous Graphics and Animations*. These were chosen because they all seem feasible

[4]Despite being designed as a tool to be used interactively in the IDE, it is possible to run EcoAndroid in *batch mode*. This is useful to process many projects, as we have done in our evaluation (see below).

to be implemented with little to none human intervention and also because they are not implemented by any of other existing refactoring tools. As described below, in some of these patterns more than one case was implemented (there are 10 cases in total). EcoAndroid provides two types of warnings: informational ones and non-informational ones. The first type does not have an automated refactoring associated. This is because either i) the suggestion is impossible to implement without further information (e.g. in the case of the *Push Over Poll* energy pattern, registration of the class in Firebase is needed) or ii) the required changes affect too much code. For these cases, if the developer wishes to follow EcoAndroid's suggestion and implement the changes manually, the plugin introduces a TODO comment so that the change is listed in the IDE's TODO window. The second type of warning has an automated refactoring associated and will change the code by applying the identified energy pattern. Figure 3 shows the 10 cases supported by EcoAndroid. Cases with gray background represent informational cases, where no source code is altered. We briefly describe all the cases below, but due to space restrictions, we only discuss in detail two cases: *Dynamic Wait Time* and *GZIP Compression*. For more details on how EcoAndroid refactors the code, the reader can consult the entries of the considered patterns on Cruz and Abreu's catalog [5].

1) Dynamic Retry Delay: The goal of the *Dynamic Retry Delay* pattern is to increase the interval between attempts to access a resource, avoiding trying to constantly access a resource that most likely went down. If an attempt to access a resource fails, the time between attempts should be increased, until a certain value, in order to spread over time the accesses. If the access is successful, the interval should not be changed. This energy pattern has two cases: *Dynamic Retry Delay* and *Check Network*. We describe in detail the first case.

a) Dynamic Wait Time: The first case of the Dynamic Retry Delay energy pattern is named, by the plugin, *Dynamic Wait Time*. The interval between threads sleep should grow exponentially and not stay constant, decreasing the chance of trying to access a resource that most likely went down. In the example shown in Listing 1, there is a `sleep` invocation, using the `backOffSeconds` variable. This variable comes from the parameter of the method `startLongPoll`. As it is a parameter, the inspection looks for method calls of the method `startLongPoll` in the current Java file. As we can observe, there is a method call which uses the `newBackOffSeconds` variable to invoke the method. The variables are assigned with constant values, either 30 or 60. When processing this example, the plugin flags this as a problem, showing up as a warning on the variable `backOffSeconds`. In this case, EcoAndroid presents the user with two warnings:

- **Warning 1: "EcoAndroid: Dynamic Retry Delay Energy Pattern - information about a new approach to implement it"**
  The informational warning shown does not alter any source code: it only adds a comment with a link that further explains how to use the `WorkRequest` class instead of using the `Thread` class. This is shown in Listing 2.
- **Warning 2: "EcoAndroid: Dynamic Retry Delay Energy Pattern - switching to a dynamic wait time between resource attempts case"**
  The second option presented to the developer alters the source code, changing every static variable assignment to a dynamic one. It starts by creating a variable entitled `accessAttempts`, initialized at 0. As the name suggests, the variable holds the number of access attempts to a resource. Then every static assignment done to the variable that puts the thread to sleep, in this case it is `newBackoffSeconds`, is altered to an incremental assignment of the `accessAttempts` variable. Finally, the number of access attempts is altered to a value of time with an upper bound. Listing 3 represents the application of the *Dynamic Wait Time* pattern, which applies the alterations described.

```
private void startLongPoll(String polledFile,
    int backOffSeconds) {
    pollingTask = new Thread () {
        public void run() {
            long start_time = System.currentTimeMillis();
            long longpoll_timeout = 480;
            int newBackoffSeconds = 0;
            if(backOffSeconds != 0) {
                log.info("Backing off for "+
                backOffSeconds + " seconds");
                try {
                    Thread.sleep((long)
                    (backOffSeconds * 1000));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            if(System.currentTimeMillis() - start_time <
            longpoll_timeout * 1000) {
                log.info("Longpoll timed out to quick,
                backing off for 60 seconds");
                newBackoffSeconds = 60;
            }
            else {
                log.info("Longpoll IO exception,
                restarting backing off {} seconds"
                + 30);
                newBackoffSeconds = 30;
            }
            startLongPoll(polledFile, newBackoffSeconds);
```

Listing 1: Dynamic Wait Time - code smell detected.

```
pollingTask = new Thread () {
    /*
     * TODO EcoAndroid
     * DYNAMIC RETRY DELAY ENERGY PATTERN INFO WARNING
     * Another way to implement a mechanism that manages
     the execution of tasks and
     their retrying, if said task fails
     * This approach uses the android.work package
     * If you wish to know more about this topic,
     read the following information:
     * https://developer.android.com/topic/libraries/
     architecture/workmanager/how-to/define-work
     */
    public void run() {...}
```

Listing 2: Dynamic Wait Time - energy pattern applied (information about a new approach to implement it).

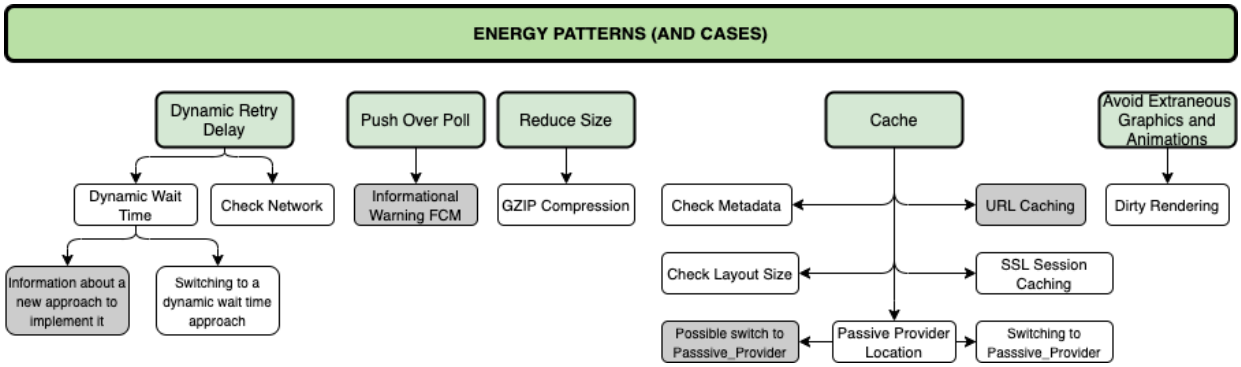

Fig. 3: Energy patterns supported by EcoAndroid. Cases with gray background represent informational cases, where no source code is altered.

```
private void startLongPoll(String polledFile,
    int backOffSeconds) {
    pollingTask = new Thread () {
        int accessAttempts = 0;
        public void run() {
            ...
            if(System.currentTimeMillis() - start_time
                < longpoll_timeout * 1000) {
                log.info("Longpoll timed out to quick,
                    backing off for 60 seconds");
                accessAttempts++;
            }
            else {
                log.info("Longpoll IO exception,
                    restarting backing off {} seconds"
                + 30);
                accessAttempts++;
            }
            newBackoffSeconds = (int) (60.0 *
                (Math.pow(2.0, (double) accessAttempts)
                - 1.0));
            startLongPoll(polledFile, newBackoffSeconds);
```

Listing 3: Dynamic Wait Time - energy pattern applied (switching to a dynamic wait time between resource attempts).

2) Push Over Poll: A *push notification* establishes and maintains a connection with a server over the Internet, allowing the server to send data to the application when something has actually changed on the server. On the other hand, *polling* is the continuous checking of other programs or devices by one program or device to check what state they are in, usually to see whether they are still connected or want to communicate. The goal of this energy pattern is to use push notifications instead of actively querying resources, such as polling. This transformation is specifically beneficial when there is not a significant number of notifications, as shown by Dinh and Boonkrong [24], who compare battery usage between these two techniques. If there is not a significant number of notifications coming in, pushing notifications will be a better choice since it's not always actively querying resources. Otherwise, the difference between these two mechanisms is not as significant. This energy pattern has one case: *Informational Warning FCM*.

3) Reduce Size: The goal of the pattern *Reduce Size* is to reduce the size of the data being transferred as much as possible, therefore reducing the energy being used in the transfer. The

```
HttpURLConnection con =
        (HttpURLConnection) url.openConnection();
System.out.println("Length : " + con.getContentLength());
Reader reader =
        new InputStreamReader(con.getInputStream());
```

Listing 4: GZIP Compression - code smell detected.

```
HttpURLConnection con =
        (HttpURLConnection) url.openConnection();
con.setRequestProperty("Accept-Encoding", "gzip");
System.out.println("Length : " + con.getContentLength());
Reader reader;
if ("gzip".equals(con.getContentEncoding())) {
    reader = new InputStreamReader(new GZIPInputStream(
                                    con.getInputStream()));
} else {
    reader = new InputStreamReader(con.getInputStream());
}
```

Listing 5: GZIP Compression - energy pattern applied.

change to be made consists in transforming/compressing the data being transmitted, whenever a data transfer occurs. This energy pattern has one case: *GZIP Compression*. The goal is to ensure that any given response from an URL Connection is compressed by the GZIP scheme. Consider the example shown in Listing 4, where there is an open connection receiving an input stream that is not requested to be compressed. For this case, EcoAndroid shows a warning and allows the developer to automatically refactor the code into the code shown in Listing 5.

4) Cache: The goal of the *Cache* pattern is to store data that is being used frequently, which means a lower energy consumption since it reduces the amount of code executed. This energy pattern has five cases: *Check Metadata*, *Check Layout Size*, *SSL Session Caching*, *Passive Provider Location* and *URL Caching*.

5) Avoid Extraneous Graphics and Animations: Graphics and animations are resources associated with an high energy consumption. The intent of the *Avoid Extraneous Graphics and Animations*'s pattern is to reduce the usage of this resources as much as possible. This is particularly relevant for any resource associated with an high energy consumption that does not have a direct impact on the user experience.

However, knowing when to apply this pattern is a challenge since it is difficult to know exactly when a resource is strictly needed or when the resource does not have a direct impact on the user experience. The energy pattern has one case: *Dirty Rendering*. EcoAndroid supports only a simple case: it is able to change the rendering mode from `GLSurfaceView.RENDERMODE_CONTINUOUSLY` to `GLSurfaceView.RENDERMODE_WHEN_DIRTY`.

IV. Evaluation

Given that the goal of EcoAndroid is to automatically apply a set of energy patterns to Java source code, we measured how many refactorings EcoAndroid suggests for a realistic set of mobile Java applications.

A. Mobile Applications Analyzed

We used Android mobile applications retrieved from *F-droid* [25], an alternative app store that catalogs over 2000 mobile applications that are Free and Open Source Software (FOSS). We retrieved meta-information about all the F-Droid applications[5] and we filtered and ordered them before being used for the evaluation. We processed information relative to 2319 mobile applications from which we filtered 1615 applications with the following characteristics:

- The source code of the application is available in GitHub;
- The GitHub project is not archived;
- The GitHub project has had a commit since 2018;
- The source code of the application is written in Java.

From the F-Droid applications retrieved, there was a total of 474 mobile applications where the main language was not Java, 89 mobile applications where the GitHub project was archived, and 141 mobile applications with the last commit made to the GitHub project more than 2 years ago. The mobile applications were then sorted by the following order: 1) Percentage of Pull Requests accepted; 2) Date of Last Commit; 3) Total merged Pull Requests; 4) Number of GitHub Stars; 5) Number of GitHub Watchers. The first three criteria were chosen to increase our chances of having feedback from developers. Our intuition is that maintainers of projects that accept more pull requests might be more open to discuss our proposals. The last two criteria were chosen to maximize impact by selecting popular projects. After filtering and ordering the mobile applications, the top 100 applications were used in the evaluation process. This corresponds to a total of 7441 Java files and 1,468,597 LOC. Table II summarizes the main characteristics of the GitHub projects, considering both processed and inspected projects.

B. EcoAndroid Refactorings

We were interested in determining how many refactorings are suggested by EcoAndroid for the top 100 mobile applications retrieved from the filtered and ordered dataset. For this, we executed EcoAndroid in batch mode, since doing it manually for 100 applications would be too time-consuming. Table III presents the results. The lines with a gray background refer to informational warnings, whose refactorings introduce TODOs into the source code. We divided this evaluation in three stages: we first processed the top twenty mobile applications, then the following twenty, and then the remaining sixty applications.

During the first stage, it became clear that an application of the *Check Network* energy pattern in the Second Screen app did not make sense, since the application did not declare permission to use the internet. We updated EcoAndroid accordingly and this was no longer a problem in the following stages. Also during the first stage, an application of the *Check Metadata* energy pattern in the Hacs mobile application could break some notifications of the application. We updated EcoAndroid to consider the case identified and this was no longer a problem in the following stages. We moved to the second stage and then to the third stage, thus analysing a total of 100 applications. In the second and third stages there were no problems identified and no changes to EcoAndroid were required.

A total of 95 refactoring opportunities were found in 35 projects, in a total of 7441 Java files, giving an average of one refactoring per $78.33 \approx 78$ files. Since, in average, the source code of a mobile application inspected has 74.41 Java files, this means an average of around $0.95 \approx 1$ refactorings per project. This is the case with most projects.

Case Analysis: The case with the most refactorings is *URL Caching* with 42.1% of the occurrences. It is followed by *Check Metadata* (14.7%), *Passive Provider Location* (11.6%), *SSL Session Caching* (10.5%), *Push Over Poll* (8.4%), and *Check Network* (5.3%). EcoAndroid found no opportunities for applying refactorings related to the cases *Dynamic Wait Time*, *Check Layout Size* and *Dirty Rendering*. The combination of patterns with the highest number of associated refactorings is *URL Caching* with *GZIP Compression* with nine projects being affected. This is expected since they both look for an invocation of the method *URLConnection#openConnection()* as a first step. The next two combinations with the most occurrences are *URL Caching* with *SSL Session Caching* and *URL Caching* with *Passive Provider Location*, both affecting three mobile applications. With two occurrences, the combination *Check Network* and *URL Caching* is next. With only one occurrence are the combinations: *Info Warning FCM* with *URL Caching*, *Info Warning FCM* and *GZIP Compression*, *Check Network* and *Check Metadata*, *Check Network* and *SSL Session Caching*, *Check Metadata* and *Passive Provider Location*, *Check Metadata* and *URL Caching* and the last one is *Check Metadata* and *GZIP Compression*.

V. Threats to Validity

There are two main aspects that may affect the validity of our work and findings. First, due to the complexity of EcoAndroid, there may exist implementation bugs. We extensively tested the tool to mitigate this risk. Moreover, since EcoAndroid was able to refactor a substantial amount of real-world code written by multiple people, we are confident in our

[5]Collection date: 25 June 2020

		GitHub Watchers	GitHub Stars	GitHub Forks	GitHub Contributors	Merged PRs	Closed PRs	% PRs Accepted
All Apps	**Min**	0	0	0	0	0	0	0
	Mean	18.53	185.37	65.27	10.21	39.71	48.20	0.61
	Max	1692	25630	8716	317	2603	3009	1
Top 100	**Min**	0	0	0	1	1	1	1
	Mean	6.3	38.44	11.61	8.43	5.88	5.88	1
	Max	29	616	181	317	60	60	1

TABLE II: F-Droid mobile applications characteristics.

Energy Patterns	Case	Refactorings
Dynamic Retry Delay	Dynamic Wait Time	0
	Check Network	5
Push Over Poll	Info Warning FCM	8
Reduce Size	GZIP Compression	14
Cache	Check Metadata	7
	SSL Session Caching	10
	Check Layout size	0
	Passive Provider Location	11
	URL Caching	40
Avoid Extraneous Graphics and Animations	Dirty Rendering	0
Total		95

TABLE III: Number of energy opportunities detected by EcoAndroid.

implementation. Furthermore, the code and raw data produced in this work are publicly available for other researchers and potential users to check the validity of the results.

Second, even though measuring the energy-saving impact of refactorings proposed by EcoAndroid is out of the scope of this work, whose main goal is to simplify the application of existing energy patterns, it must be stated that there is the risk that a proposed refactoring does not improve energy efficiency, since EcoAndroid does not perform energy profiling. Even if EcoAndroid is extended with existing energy profiling techniques, it might be difficult to provide strong guarantees: as documented by Ahmad et al [26], accurate mobile application energy profiling is a non-trivial task, particularly when using software-based energy profiling techniques (we argue that hardware-based energy profiling techniques, despite generally being more accurate, are not suitable for this context due to their lack of scalability).

To mitigate this risk, we have followed closely the research community proposals, focusing on Cruz and Abreu's catalog [5]. The fact that Cruz and Abreu analysed commits, issues and pull requests from a large number of applications (1021 Android apps and 756 iOS apps) to identify design practices that improve energy efficiency, and produced a catalog based on 1563 energy-related changes of mobile apps, significantly increases the confidence on the energy efficiency of the refactorings proposed by EcoAndroid.

VI. Conclusion

EcoAndroid is an Android Studio plugin that automatically applies a set of energy patterns to Java source code. We used EcoAndroid to analyze 100 Java mobile applications from F-Droid ($\approx$ 1.5M LOC) and we found that 35 of the projects had a total of 95 energy code smells detected by the plugin. EcoAndroid was able to automatically refactor all the code smells identified. These results suggest that EcoAndroid is useful and is capable of applying energy patterns to real-world mobile applications. Since the scripts that collect and process applications from F-Droid are also distributed with EcoAndroid, we argue that our artefacts are valuable assets for driving reproducible research in automated energy optimization of mobile applications.

Summary of Findings: The development of EcoAndroid and its evaluation gave us several insights. The two most important are:

1) There is a significant number of opportunities to apply energy patterns in real-world Java Android applications. When considering 100 applications taken from the F-Droid repository, EcoAndroid found a total of 95 refactoring opportunities in 35 projects. This suggests that there are many Java Android applications that could benefit from the use of EcoAndroid.
2) From the set of patterns considered by EcoAndroid, the cases with more refactorings are *URL Caching* and *Check Metadata*, with 42.1% and 14.7% of occurrences, respectively. The combination of patterns with the highest number of associated refactorings is *URL Caching* with *GZIP Compression* (9 out of 100 projects).

Future Work: Some suggestions for immediate future work include submitting refactorings proposed by EcoAndroid to project mantainers (we have started this process and preliminary results are positive); performing user studies to assess the usefulness and usability of EcoAndroid; supporting the remaining energy patterns of Cruz and Abreu's catalog [5]; and performing experiments involving energy profiling. Android Studio's energy profiler [27] is an obvious choice that can be explored, but ANEPROF [28] could also be a good option since it is a real-measurement-based power profiling tool specific for Android. Another possible direction is to extend EcoAndroid so that it supports applications written in Kotlin (the Kotlin language is supported by the PSI API).

ACKNOWLEDGMENTS

This work has been supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects UIDB/50021/2020 and PTDC/CCI-COM/29300/2017.

REFERENCES

[1] C. Wilke, S. Richly, S. Götz, C. Piechnick, and U. Aßmann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. IEEE, 2013, pp. 134–141.

[2] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 425–436.

[3] E. Noei, F. Zhang, and Y. Zou, "Too many user-reviews, what should app developers look at first?" *IEEE Transactions on Software Engineering*, 2019.

[4] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*. ACM, 2009, pp. 280–293.

[5] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, pp. 1–27, 2019.

[6] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for energy efficiency: a reflection on the state of the art," in *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. IEEE Press, 2015, pp. 29–35.

[7] M. Gottschalk, J. Jelschen, and A. Winter, "Saving energy on mobile devices by refactoring." in *EnviroInfo*, 2014, pp. 437–444.

[8] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 46–57.

[9] D. Li and W. G. Halfond, "An investigation into energy-saving programming practices for Android smartphone app development," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014, pp. 46–53.

[10] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha, "Code smells in iOS apps: How do they compare to Android?" in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 110–121.

[11] L. Cruz, R. Abreu, and J.-N. Rouvignac, "Leafactor: Improving energy efficiency of Android apps via automatic refactoring," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 205–206.

[12] L. Cruz and R. Abreu, "Using automatic refactoring to improve energy efficiency of Android apps," *arXiv preprint arXiv:1803.05889*, 2018.

[13] M. Couto, J. Saraiva, and J. P. Fernandes, "Energy refactorings for Android in the large and in the wild," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 217–228.

[14] "Autorefactor," http://autorefactor.org/, accessed: 2020-12-01.

[15] "AEON: Automated android energy-efficiency inspection," https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection, accessed: 2020-12-01.

[16] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2017.

[17] E. Iannone, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Refactoring Android-specific energy smells: A plugin for Android Studio," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 451–455.

[18] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for Android developers," in *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, vol. 2014, 2014.

[19] O. L. Goaër, "Enforcing green code with Android lint," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2020, pp. 85–90.

[20] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of Android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 115–126.

[21] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of Android applications along their evolution," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 236–247.

[22] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of Android-specific code smells: The aDoctor project," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 487–491.

[23] "Program structure interface (PSI)," https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html, accessed: 2020-12-01.

[24] P. C. Dinh and S. Boonkrong, "The comparison of impacts to Android phone battery between polling data and pushing data," in *IISRO Multi-Conferences Proceeding. Thailand*, 2013, pp. 84–89.

[25] "F-droid," https://f-droid.org, accessed: 2020-12-01.

[26] R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. Shiraz, "A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues," *Journal of Network and Computer Applications*, vol. 58, pp. 42–59, 2015.

[27] "Inspect energy use with energy profiler," https://developer.android.com/studio/profile/energy-profiler, accessed: 2020-12-01.

[28] Y.-F. Chung, C.-Y. Lin, and C.-T. King, "Aneprof: Energy profiling for android Java virtual machine and applications," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 372–379.